在 HCS08 系列 MCU 上
用软件实现仪表步进电机的驱动

飞思卡尔半导体成都分公司
傅志强（frank.fu@freescale.com）

步进电机由于具有角位移和输入脉冲数成正比并且没有累积误差的特点，而被广泛的用作汽车仪表的显示部件，其中具有代表性的是伟力驱动技术（深圳）有限公司的VID29系列步进电机。在多数情况下，人们会使用专用的驱动芯片来驱动步进电机，主控制器只需要给出方向控制信号和控制转动步数的脉冲就行了。另外，也有集成了步进电机驱动电路的MCU，如Freescale的MC9S12HY系列，其使用方法也比较简单。然而，在很多较低端的汽车仪表，如微型车、农用车、三轮货车和摩托车的仪表上，人们为了降低成本，希望能够不用专用驱动芯片或相对较贵的带驱动电路的MCU，而是用普通的MCU直接去驱动步进电机。本应用笔记介绍了在Freescale的HCS08系列MCU上，如何用软件来实现对VID29系列步进电机的直接驱动。本文所附带的程序，已经在Freescale的LG32 Cluster Reference Design演示板上运行验证过。

VID29 系列步进电机的工作原理

VID29系列步进电机是两相步进电机经三级齿轮减速传动输出的。该步进电机的工作原理可以用下面的简化的结构图（图1～4）进行说明。在象VID29系列这样的两相步进电机中，转子是一个永磁体，定子上安装了两组线圈。当给定子线圈通上电流的时候，就在转子周围的气隙中产生了一个磁场，转子就会在磁力的作用下转动到使它自身的磁场方向和线圈电流产生的气隙磁场方向平行的位置（下文中把它叫做平衡位置）。要让步进电机连续地旋转，可以按如下步骤进行：

1. 如图1所示，在线圈A中通上电流，转子就会转过90度到图2所示的位置；

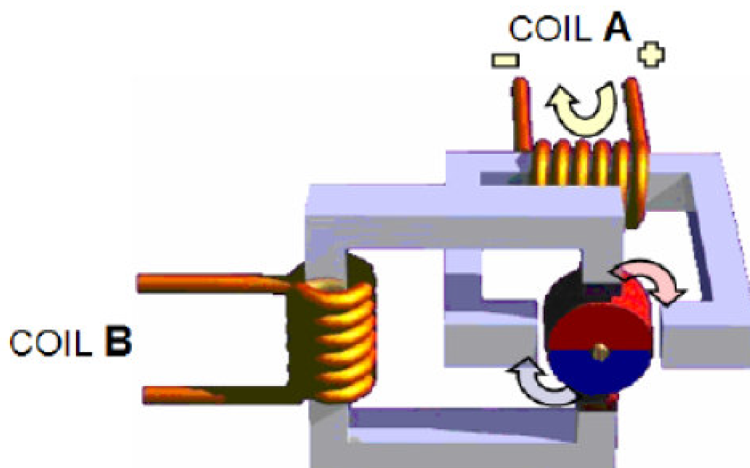

图1. 步进电机线圈A通电

2. 如图2所示，断开线圈A中的电流，给线圈B通上电流，转子又会继续旋转90度到图3所示的位置；

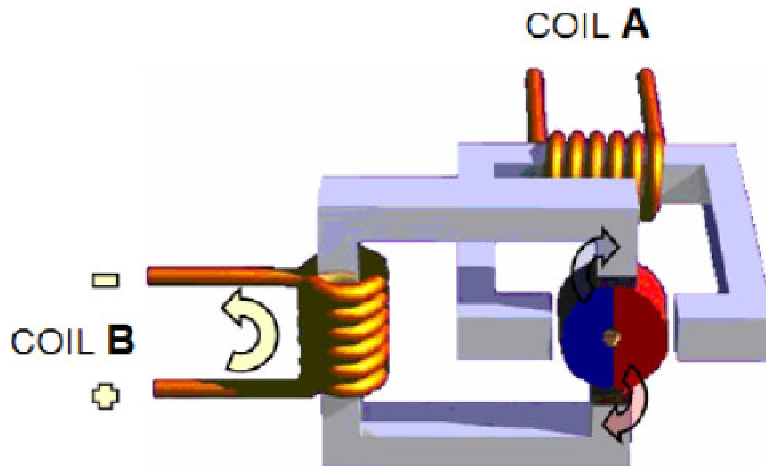

图2. 步进电机线圈B通电

3. 如图3所示，断开线圈B中的电流，给线圈A通上跟步骤1中方向相反的电流，转子继续旋转90度到图4所示的位置；

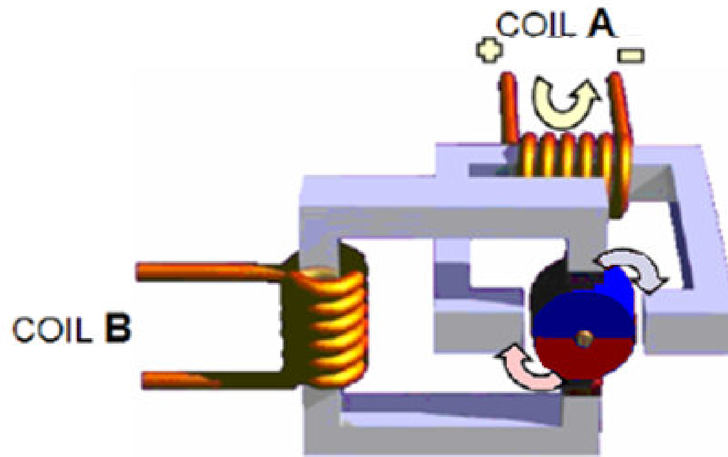

图3. 步进电机线圈A反向通电

4. 如图4所示，断开线圈A中的电流，给线圈B通上跟步骤2中方向相反的电流，转子继续旋转90度回到图1所示的位置；

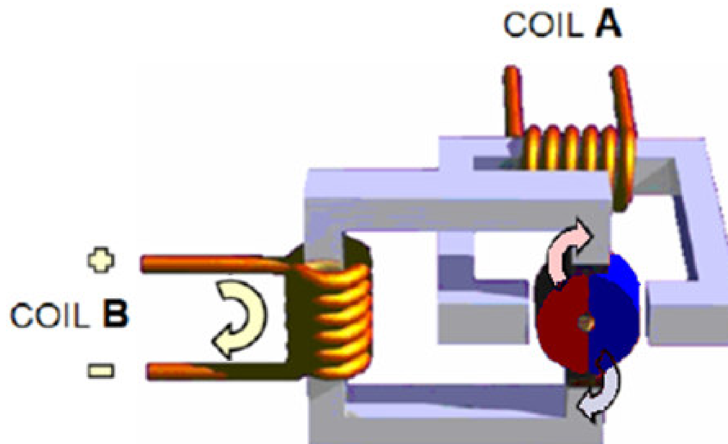

图4. 步进电机线圈B反向通电

5. 重复步骤1～4，步进电机就会连续地旋转起来了。

如果将步骤1～4的顺序颠倒过来，那么步进电机就会以相反的方向旋转。另外，如果将A和B两个线圈同时通以大小相同的电流，那么产生的合成磁场的方向就和一个线圈单独通电时的磁场方向成45度夹角，这样转子就将旋转45度，而不是90度。

实际的步进电机由于转子的形状和定子线圈的安装方位跟上述的简化结构不同，所以工作时驱动电流的时序和转子每一步旋转的角度也不完全一样。对于VID29系列步进电机来说，它的驱动脉冲序列和转子相应的旋转角度如图5所示。

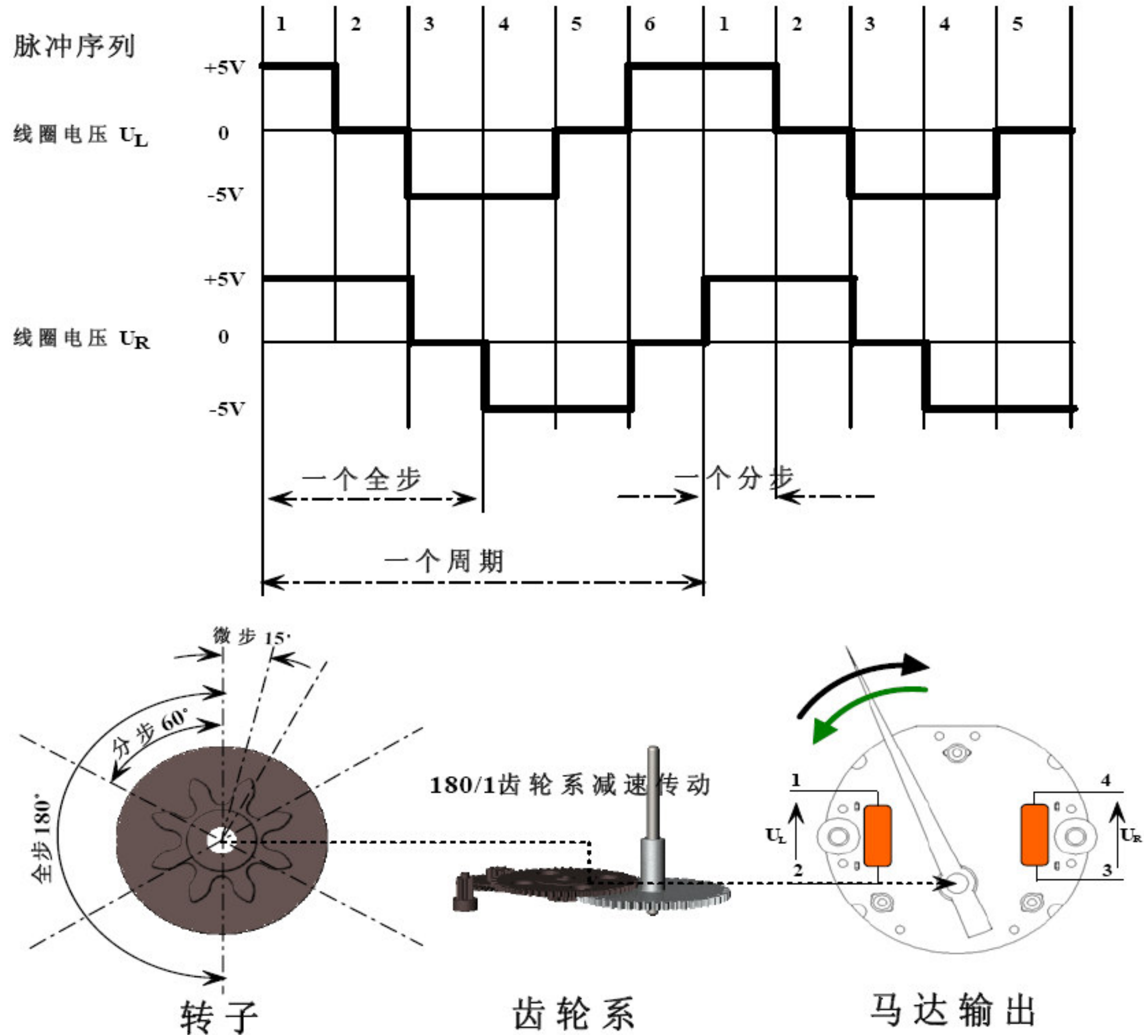

图5. VID29系列步进电机工作原理

分步驱动和微步驱动

步进电机的驱动通常有分步方式和微步方式两种，图5中的脉冲序列是分步方式下的驱动信号。分步方式的优点是驱动信号的幅度只有0和电源电压（5V）两种，与数字信号的低电平和高电平完全一一对应，因此只要使用MCU的普通数字I/O口(GPIO)就可以产生驱动信号，驱动程序也比较简单。但是由于分步方式下，定子线圈产生的气隙磁场的方向在每走一步的过程中都发生一个较大的跳变——对于VID29系列步进电机而言是60度的跳变，定子磁场从原来的方向跳变到下一个平衡位置的方向，转子则在磁力的作用下加速向

下一个平衡位置转动，当它到达平衡位置的那个瞬间，速度达到最大值，而磁场力则变为零（只考虑切向力，不考虑径向力，因为径向力与转动无关。下同）；然后，如果驱动信号没有变化的话，转子就会在惯性的作用下继续往前转动偏离平衡位置，这时磁场力将会增大，而其方向却变成跟刚才相反，于是就使转子减速，当转子与平衡位置的偏离达到最大时，其速度减为零，然后在磁场力作用下往回加速转动；当转子转回平衡位置时，磁场力又变为零，而速度不为零，于是在惯性的作用下继续转动偏离平衡位置……如此反复，只要驱动信号还没有再次改变，转子就会象荡秋千一样在平衡位置附近来回振荡，并在摩擦力的作用下幅度逐渐减小。这种振荡还会产生一定的噪声，所以用分步方式驱动时，步进电机的噪声和抖动会比较大。

为了减小步进电机运行时的噪声和抖动，人们设法让定子线圈的磁场方向的跳变幅度变小，把一个分步一次的大跳变分成若干次较小的跳变来完成，于是就有了微步驱动方式，也叫细分驱动方式。

根据矢量合成的原理，当步进电机中的两个线圈各自产生的磁场的强度按照正/余弦规律变化的时候，它们的合成磁场的方向就会匀速旋转，而合成磁场的强度保持不变，如图6所示。线圈产生的磁场强度与通过它的电流大小成正比，因此微步驱动方式就是让通过线圈的驱动电流不是象分步方式那样在0和最大值之间跳变，而是按照正/余弦规律分成几个阶梯逐步变化，如图7所示。

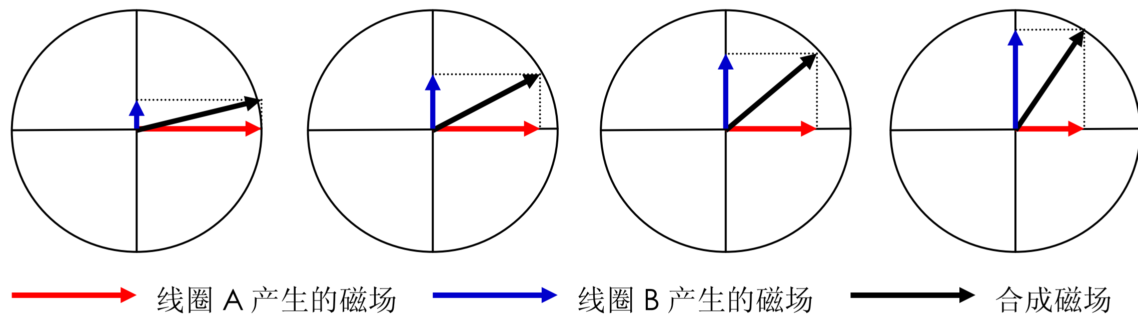

图6. 两个线圈产生的合成磁场

那么如何来产生阶梯变化的驱动电流呢？通常的做法是将一个PWM波形电压信号施加到线圈上；同时串联一个阻值较小的电阻作为电流传感器，将电流信号转换成电压信号反馈回PWM控制器中；PWM控制器根据反馈信号调整输出脉冲的占空比，从而使线圈上的平均电流等于所需的阶梯电流。目前，常见的步进电机专用驱动芯片就是依据这个原理来工作的。在使用MCU直接驱动步进电机的时候，因为MCU内部也集成了PWM模块，所以我们也可以采用这个方法。但是，这种方法还需要使用ADC来测量反馈信号的大小，并且要实时地计算出所需的PWM脉冲的占空比，因此将会占用太多的MCU的资源，使MCU几乎无法再处理其他的事情。

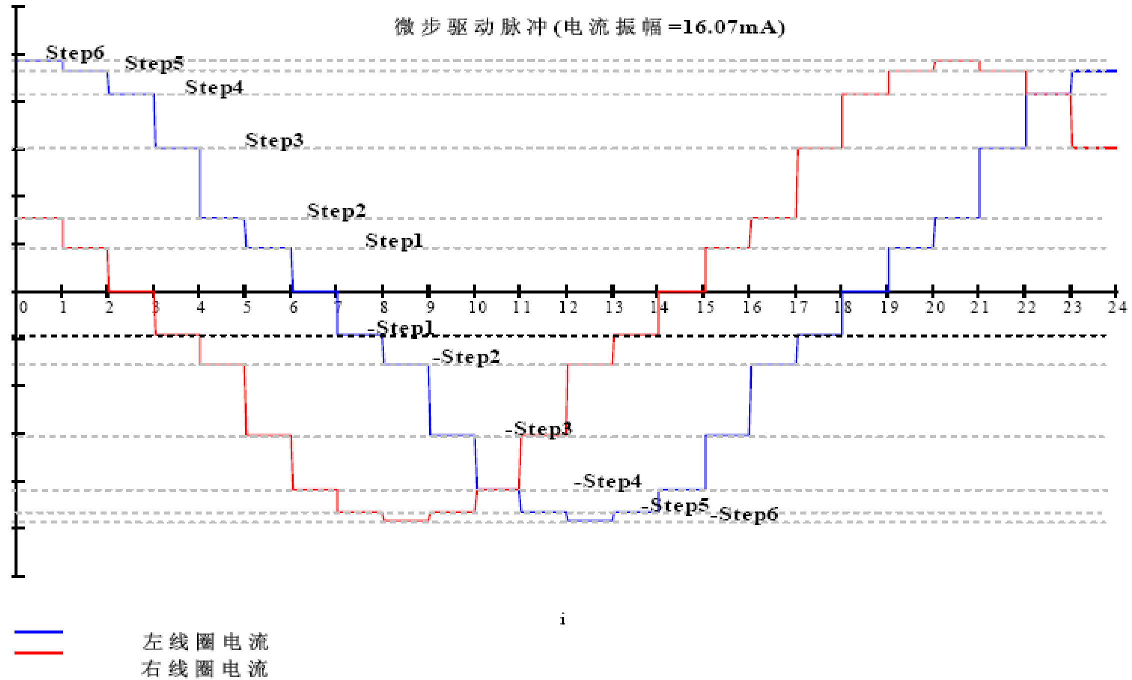

图7. 微步方式下的驱动电流

为了减少对MCU资源的占用，我们可以将其中的反馈环节删掉，从闭环控制变为开环控制。对每个阶梯电流，我们事先计算出所需的PWM脉冲的占空比，并做成查找表存放在FLASH中。在驱动步进电机转动的时候，每走一个微步就从表中取出相应的占空比来对PWM模块进行设置，然后在当前的这一步结束之前不再改变PWM脉冲的占空比。这样一来就极大的减少了计算量，使MCU有足够的资源去处理其他的事情。然而，由于定子线圈具有一定的电感，所以通过线圈的平均电流和PWM脉冲的占空比并不是线性关系，要想精确的计算出每个阶梯电流对应的占空比是相当困难的；另外，当转子旋转的时候，还会在定子线圈中产生感生电动势，这个感生电动势会使通过线圈的电流发生改变，而它的大小又与转子永磁体的磁场强度和旋转速度相关。因此，事实上在开环控制中不管你使用多么复杂的数学公式来计算占空比，都无法使通过线圈的平均电流总是等于预期的阶梯电流。不过，所幸的是：在低端的汽车仪表中，并不要求对步进电机的转动控制到那么精确的程度，我们可以姑且认为通过线圈的平均电流是和PWM脉冲的占空比成正比的，反正只要减小了每一步跳变的角度（相对于分步方式而言），就可以让步进电机转动得更平稳一些了，噪声也能减小一些。

电机转动的加减速

步进电机在转动时，因为转子、传动齿轮和负载的转动惯量，使它从一个位置转动到下一个位置（一个分步或微步）需要一定的时间。如果在转子转动到下一个位置之前，驱动信号就又往前走了一步的话，那么转子的磁场方向和定子线圈产生的气隙磁场方向的夹角就会超过一个分步或微步所对应的角度。只要转子的旋转速度跟不上驱动信号的变化速度，这个夹角就会越来越大，当夹角超过180度的时候，磁场对转子的作用力的方向就会变得跟原来的方向相反，如图8所示。这时，转子的旋转速度就会减慢，直到变成以相反的方向旋转。最终的结果就是转子所转过的角度和气隙磁场所转过的角度不相等了，也就

是转子所转过的步数和驱动信号走过的步数不相等了，人们常把这种现象叫做“失步”。同样的，当要使步进电机从高速的旋转中停下来的时候，如果驱动信号的变化过快，转子就有可能在惯性的作用下继续旋转超过180度，从而也产生失步。另外，由于转子轴承、传动齿轮和负载上都有一定的摩擦阻力，因此电机在连续转动时的速度也是有限的，如果驱动信号的变化速度超过了电机能达到的最大转速的话，电机也会失步。

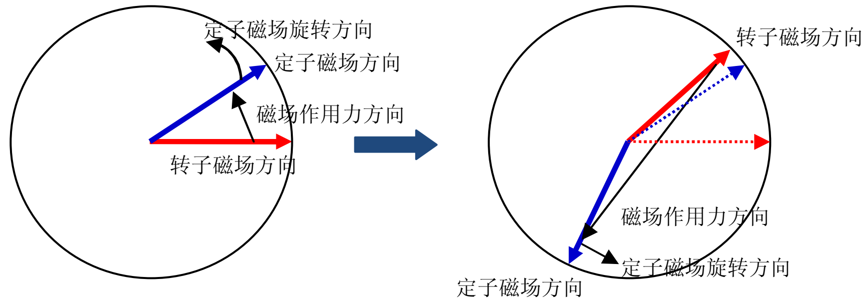

图8. 步进电机失步的过程

那么，如果让驱动信号一直保持较慢的变化速度，是不是就没有问题了呢？答案当然是否定的。这是因为步进电机作为仪表的显示部件，我们要求它能够将被测信号的变化实时地显示出来；而电机转动如果比较慢，那么仪表的显示就无法跟上被测信号的变化。

为了让步进电机既不会失步，又能转得尽可能的快，那么就要让驱动信号的变化速度和转子转动的速度保持基本一致。当电机启动的时候，转子做加速转动，这时第一步的持续时间要比较长，然后每一步的持续时间逐渐变短，对应的转动速度变化如图9所示。电机停止的过程则与之相反。在VID29系列步进电机的数据手册上给出了电机启动/停止时允许的驱动信号的变化速度（启动频率，Start-stop Frequency）和电机连续转动时允许的驱动信号的变化速度（最大驱动频率，Max Driving Frequency），我们可以根据它们计算出第一步的持续时间和加速过程结束后的每一步的持续时间。

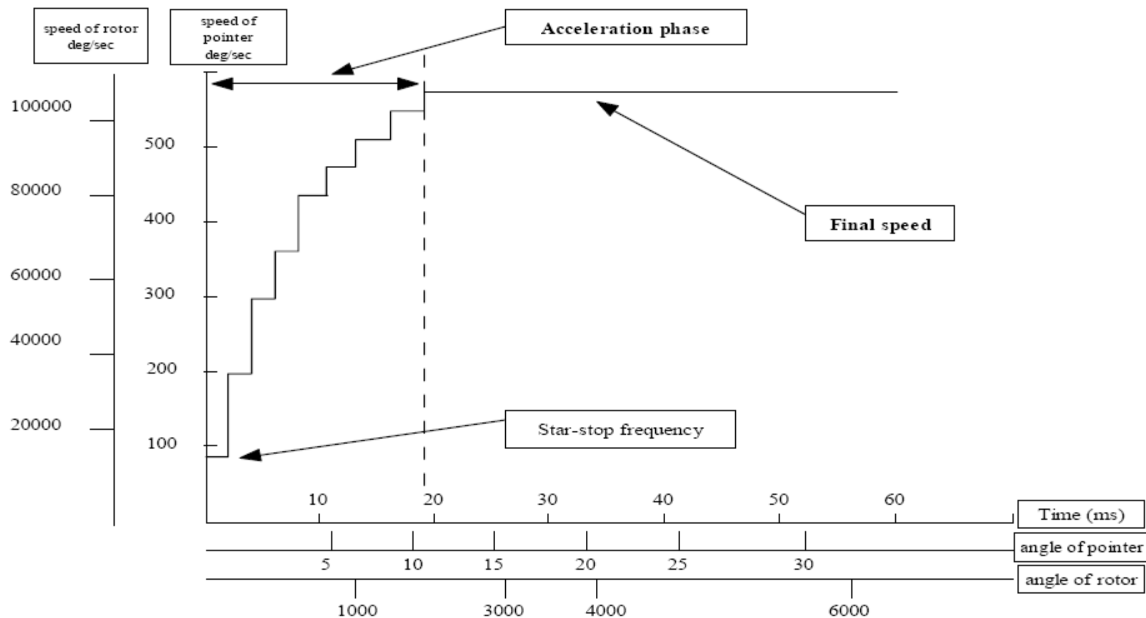

图9. 步进电机加速过程

步进电机和MCU的硬件连接

在LG32 Cluster Reference Design中，MCU和步进电机之间的连接如图10所示。其中使用了一片74ACT125作为电流放大驱动，这是因为MC9S08LG32的I/O口输出的电流最大只有10mA，而VID29步进电机需要的驱动电流最大可达20mA。使用TPM模块的两个PWM输出通道驱动步进电机两个线圈的正极，两个普通I/O口驱动两个线圈的负极。TPM模块的两个通道也可以设置成普通I/O口，这样就可以根据需要使用微步方式驱动或者分步方式驱动。

在MC9S08LG32中集成了两个TPM模块，其中TPM1有2个通道，TPM2有6个通道。在这里选择TPM1来驱动步进电机，是因为当把一个TPM模块的某一个通道设置为PWM输出时，此TPM模块公用的模数(MOD)寄存器将被设成一个比较特殊的值，这样就会给它的其它通道的功能使用造成很多限制。所以为了更加充分地利用MCU的资源，这里选择了通道较少的TPM1，而把通道较多的TPM2留作他用。

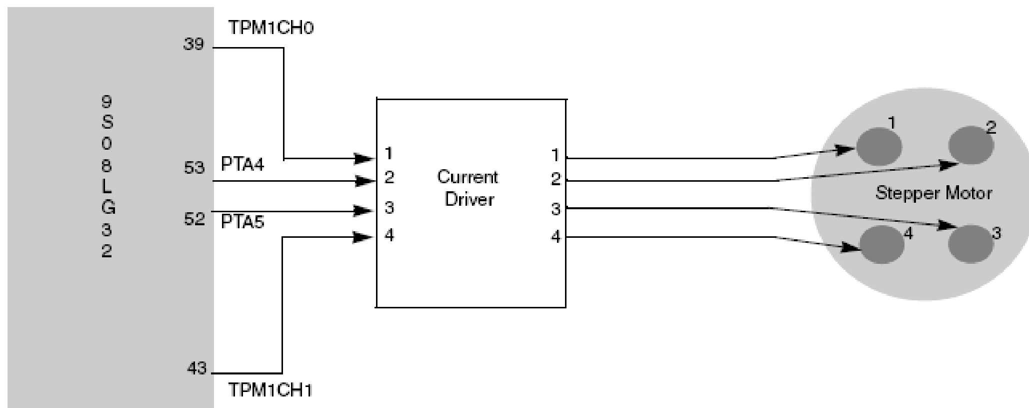

图10. MCU和步进电机连接图

步进电机驱动程序

本驱动程序为应用程序提供的接口函数有三个：

1. InitStepper：初始化函数，对驱动所用到的I/O口和定时器进行设置，并让步进电机转到初始位置——使仪表的指针指到零点的位置。
   函数原型：void InitStepper(void);
   参数：无
   返回值：无
2. SetStepperTarget：设定步进电机的目标位置，也就是要让仪表的指针转到哪个位置（以相对于零点的角度来表示），整个转动的过程（如需要转多少步，往哪个方向转等）应用程序无需知道，而且应用程序可以指定任意目标位置。
   函数原型：void SetStepperTarget(word wTargetDegree);
   参数：word wTargetDegree，目标位置相对于零点的角度，其值是以(1/12)度（等于微步方式下的步距）为单位的；
   返回值：无
3. GetStepperCurrent：得到步进电机的当前位置。
   函数原型：word GetStepperCurrent(void);
   参数：无

返回值：返回仪表的指针当前所指的位置（以相对于零点的角度来表示），其值也以(1/12)度（等于微步方式下的步距）为单位。

另外，在头文件Stepper.h中定义了选择驱动方式的宏，使用者修改宏定义就可以选择分步驱动方式或微步驱动方式，如下：

```
#define STEPPER_PARTIAL_MODE     0
#define STEPPER_MICRO_MODE       !STEPPER_PARTIAL_MODE
```

如前所述，驱动信号的变化速度不能太快，每次变化后都必须保持一段时间不变，为了让驱动程序占用的CPU的处理时间更少，就需要用到一个定时器。除了上面的三个接口函数外，驱动程序的其余部分都在定时器的中断服务程序中运行。

在驱动程序中，用两个静态变量来保存步进电机的目标位置和当前位置，其值是到零点的步数。定时器中断由SetStepperTarget函数使能，在中断服务程序中根据目标位置和当前位置的差值来决定步进电机转动的方向和步数，然后根据选择的驱动方式执行相应的转动程序。

分步方式程序

从前面的图5中，可以看到分步方式下的驱动信号是周期性变化的，每个周期对应转子旋转一圈（输出轴旋转2度），其中包含6个分步，分别对应转子的6个平衡位置，我们也可以把它们叫做6个相位。因此，分步驱动方式下MCU的4个I/O口输出的电平也有6种组合，分别对应这6个相位。

在程序中，使用一个静态局部变量bPhase来保存步进电机当前所处的相位，步进电机每走一个分步，相位就加1（逆时针转）或者减1（顺时针转），然后在一个switch…case结构中按新的相位设置I/O口的输出状态，如下：

```
switch(bPhase)
 {
     case 0:
     STEPPER_LP = 1;
     STEPPER_LN = 0;
     STEPPER_RP = 1;
     STEPPER_RN = 0;
     break;
     case 1:
     STEPPER_LP = 0;
     STEPPER_LN = 0;
     STEPPER_RP = 1;
     STEPPER_RN = 0;
     break;
     case 2:
     STEPPER_LP = 0;
     STEPPER_LN = 1;
     STEPPER_RP = 0;
     STEPPER_RN = 0;
     break;
     case 3:
     STEPPER_LP = 0;
     STEPPER_LN = 1;
     STEPPER_RP = 0;
     STEPPER_RN = 1;
```

```
        break;
        case 4:
        STEPPER_LP = 0;
        STEPPER_LN = 0;
        STEPPER_RP = 0;
        STEPPER_RN = 1;
        break;
        case 5:
        STEPPER_LP = 1;
        STEPPER_LN = 0;
        STEPPER_RP = 0;
        STEPPER_RN = 0;
        break;
        default:
        ;
    }
```

微步方式程序

从前面的图7中，可以看到微步方式下的驱动信号是近似于正弦波的阶梯电流信号，每个周期对应转子旋转一圈（输出轴旋转2度），当把每个分步细分成4个微步的时候，一个正弦波周期里就有24个阶梯。本文利用PWM来产生这些阶梯电流，并且近似地认为PWM的占空比与平均电流大小成正比。

首先，把TPM1的通道0和1初始化成边沿对齐的PWM工作模式，设置TPM1的工作时钟为总线时钟，预分频系数为1，设置TPM1MOD的值为255，这样当总线时钟频率为16MHz时，所产生的PWM信号的频率是64KHz。在对TPM进行设置的时候需要考虑两个方面：

其一，因为PWM的占空比与平均电流大小成正比，所以要使产生的平均电流尽量接近理想的正弦波分成24个阶梯时的值，就要求PWM占空比的精度比较高。TPM1MOD的值决定了每个PWM周期里有多少个时钟周期，脉冲宽度则由各个通道的Value寄存器的值决定，最小为0（对应占空比为0），最大为(TPM1MOD+1)（对应占空比为100％），设置Value寄存器的值大于(TPM1MOD+1)的效果和设为等于(TPM1MOD+1)是一样的。PWM的占空比其实就是Value寄存器的值和(TPM1MOD+1)的比值，由于寄存器里的值只能是整数，所以占空比的精度只能达到1/(TPM1MOD+1)。因此，要使占空比的精度较高，就要给TPM1MOD设置比较大的值。

其二，因为电机的线圈是个感性负载，所以用PWM驱动的时候，线圈上的电流是波浪式变化的，如图11所示。电流的变化导致磁场强度的变化，两个线圈各自产生的磁场强度变化则导致合成磁场的强度和方向都发生变化，这种波浪式的变化将使电机产生振动和噪声。显然，电流波纹的幅度是PWM信号的周期的单调增函数，即PWM的周期越长则电流波纹的幅度越大。因此，PWM的周期应当尽量短，而PWM的周期等于(TPM1MOD+1)乘以TPM1的工作时钟的周期，所以应当使TPM1的工作时钟的频率尽量高，而且TPM1MOD的值不宜设得太大。

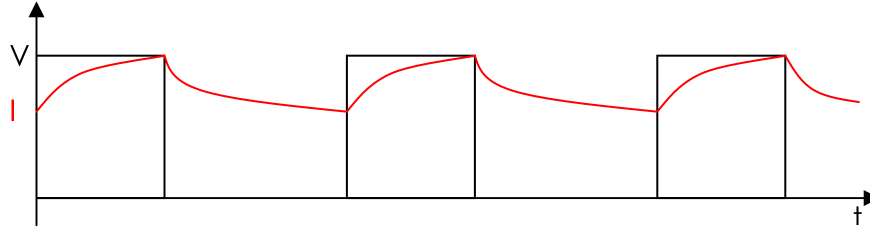

图11. PWM驱动信号和相应的线圈电流

设置完PWM的工作时钟频率和模值（TPM1MOD）后，还要计算出各个阶梯电流所对应的占空比，即Value寄存器的值，计算公式如下：

$$V = (TPM1MOD+1)\times\cos(\frac{i}{24}2\pi)$$

$$= 256\times\cos(\frac{i}{24}2\pi)\ ,\ i=0,1,2,\cdots\cdots 23$$

把算出的数值做成查找表放在FLASH存储器中。两个线圈的电流相位相差60度（$\pi/3$），而$\cos(\frac{i}{24}2\pi-\frac{\pi}{3})=\cos(\frac{i-4}{24}2\pi)$，因此把由上式计算得到的一组数字的顺序循环移动4个位置就得到另一个线圈的电流对应的查找表。由于PWM的占空比只能控制平均电流的大小，而电流的方向由其他设置决定，所以程序中这两个表的数字都取绝对值，它们的定义如下：

```
static const word waCosineTable[] =
{
    256, 246, 221, 180, 127, 65, 0, 65, 127, 180, 221, 246,
    256, 246, 221, 180, 127, 65, 0, 65, 127, 180, 221, 246
};
static const word waShiftCosTable[] =
{
    127, 180, 221, 246, 256, 246, 221, 180, 127, 65, 0, 65,
    127, 180, 221, 246, 256, 246, 221, 180, 127, 65, 0, 65
};
```

与分步方式程序相似地，本文把这24个阶梯叫做24个相位，用一个静态局部变量bPhase来保存步进电机当前所处的相位，步进电机每走一个微步，相位就加1（逆时针转）或者减1（顺时针转），然后在两个查找表中取出相应的数值写到TPM1的两个通道的Value寄存器中。为了实现电流的方向变化，需要在电流过零对应的相位时改变PWM输出的极性和连接线圈另一极的GPIO口输出的电平，在程序中用switch…case结构来完成这个任务，代码如下：

```
void MoveStepper1Micro(void)
{
    static byte bPhase;

    if(bDirection == 1)
    {
        if(bPhase == 0)
            bPhase = 23;
        else
            bPhase--;
        switch(bPhase)
        {
            case 17:
```

```
            STEPPER_LN = 1;      //设置左线圈负极GPIO口输出高电平，并
            TPM1C0SC = 0x24;     //设置PWM极性为低有效，使电流方向为负
            break;
            case 21:
            STEPPER_RN = 1;      //设置右线圈负极GPIO口输出高电平，并
            TPM1C1SC = 0x24;     //设置PWM极性为低有效，使电流方向为负
            break;
            case 5:
            STEPPER_LN = 0;      //设置左线圈负极GPIO口输出低电平，并
            TPM1C0SC = 0x28;     //设置PWM极性为高有效，使电流方向为正
            break;
            case 9:
            STEPPER_RN = 0;      //设置右线圈负极GPIO口输出低电平，并
            TPM1C1SC = 0x28;     //设置PWM极性为高有效，使电流方向为正
            break;
            default:
            ;
        }
        wCurrentPosition++;
    }
    else if(bDirection == 0xFF)
    {
        if(bPhase == 23)
            bPhase = 0;
        else
            bPhase++;
        switch(bPhase)
        {
            case 6:
            STEPPER_LN = 1;      //设置左线圈负极GPIO口输出高电平，并
            TPM1C0SC = 0x24;     //设置PWM极性为低有效，使电流方向为负
            break;
            case 10:
            STEPPER_RN = 1;      //设置右线圈负极GPIO口输出高电平，并
            TPM1C1SC = 0x24;     //设置PWM极性为低有效，使电流方向为负
            break;
            case 18:
            STEPPER_LN = 0;      //设置左线圈负极GPIO口输出低电平，并
            TPM1C0SC = 0x28;     //设置PWM极性为高有效，使电流方向为正
            break;
            case 22:
            STEPPER_RN = 0;      //设置右线圈负极GPIO口输出低电平，并
            TPM1C1SC = 0x28;     //设置PWM极性为高有效，使电流方向为正
            break;
            default:
            ;
        }
        wCurrentPosition--;
    }
```

```
    TPM1C0V = waCosineTable[bPhase];
    TPM1C1V = waShiftCosTable[bPhase];
}
```

转动速度控制

由于被测量的输入信号的变化是随机的，有时快有时慢，为了让仪表实时、直观地显示出输入信号的变化，驱动程序就必须对步进电机的转动速度进行适当的控制。可以说，一个汽车仪表做得好不好，很大程度上就看它对步进电机的转动控制得好不好。

通常，仪表对输入信号进行周期性的定时测量，因此对步进电机目标位置（指针位置）的设定也是每隔一段时间进行一次的。在程序中给这个间隔时间定义了一个宏，放在Stepper.h头文件中，以便根据应用程序的需要作修改，宏定义如下：

```
#define STEPPER_UPDATE_INTERVAL 256     //单位：毫秒，必须在(8,512)内
```

这里把指针位置的更新间隔时间限定在8到512毫秒之间是因为：

1. 仪表是用来给人看的，而人眼并不能看清变化非常快的东西，所以对指针位置更新得太快就没有实际意义，徒然增加CPU的处理负担而已。
2. 如下文将要提到的，本程序使用的定时器所能达到的最大中断间隔时间为512毫秒，为了简化定时器的中断服务程序，减小CPU的负荷，所以限定指针位置的更新间隔时间不要超过512毫秒。

为了让仪表的指针摆动得更加平稳、自然，最好能让电机在下一次设定目标位置前的那一刻刚好转动到本次设定的目标位置。为此就要计算出电机在这一段时间里的平均旋转速度，也就是每个分步或微步的平均间隔时间。当所需的平均速度较大时，如前所述，还需要对电机进行加减速的控制。

为了实现步进电机的转动速度控制，在定时器的每次中断里只让电机转动一个分步/微步，然后用一定的算法计算出这一步到下一步之间应当间隔的时间，以重新设置定时器的溢出时间。在本驱动程序中，定时器选用的是TPM2模块的通道0，对其相关的寄存器的设置说明如下：

初始化：

```
    SCGC1_TPM2 = 1;      //使能TPM2模块的时钟
    TPM2MOD = 0;       //设置TPM2的计数器为自由运行模式
    TPM2SC = 0x0F;     //选择总线时钟为时钟源，预分频系数为128
```

把预分频系数设置为128是为了让定时器的溢出时间能够尽可能大，减少中断对CPU造成的负担。当总线时钟频率为16MHz时，此定时器能达到的最大中断间隔时间是512毫秒。

启动定时器：

```
    TPM2C0V = TPM2CNT + 2;
    TPM2C0SC = 0x50;
```

设置通道0为输出比较模式，比较一致时产生中断，但比较结果不输出到管脚上（相关管脚仍然由GPIO控制）。因为启动定时器时步进电机还没有开始转动，所以给比较值寄存器(TPM2C0V)设置的值是计数器(TPM2CNT)当前的值加2，即让它尽快地产生一次中断。然后在每次中断产生后，在中断服务程序里让步进电机转动一步，并给TPM2C0V设置一个新的值，即根据所需的间隔时间给TPM2C0V增加一定的数值。

为了对步进电机进行加减速控制，我们需要知道它启动时第一步所需的间隔时间和最大转速时每一步所需的间隔时间。步进电机启动时第一步所需的间隔时间的最小值(Tss)可以由下面的公式计算得到：

$$Tss = \frac{d}{fss}$$

式中，d是步距，即每走一步输出轴转过的角度；fss是启动频率，即启动时允许的输出轴的最大转速，单位是（度/秒）。

从VID29系列步进电机的数据手册上可以查到，fss等于125(度/秒)，在分步方式下d等于(1/3)度。代入上式，可以算出Tss等于(1/375)秒，约等于2667微秒。

在程序中定义MAX_PARTIAL_STEP_TIME为400，当在中断服务程序里把TPM2C0V的值增加400后，下一次比较中断产生的时间和本次中断之间的间隔就为3200微秒，大于Tss且还有一定的裕量。

步进电机达到最大转速时每一步所需的间隔时间(Tmm),计算公式如下：

$$Tmm = \frac{d}{fmm}$$

式中，d仍是步距，fmm是最大驱动频率，即连续旋转时输出轴的最大转速，单位是（度/秒）。

从VID29系列步进电机的数据手册上查到fmm等于600(度/秒)，算出Tmm约等于556微秒。在程序中定义MIN_PARTIAL_STEP_TIME为80，对应的间隔时间为640微秒，同样是大于Tmm且有一定的裕量。

对于微步方式，由于我们把每个分步细分成4个微步，所以相应的Tss和Tmm都是分步方式时的四分之一。

在程序中，设定步进电机的目标位置SetStepperTarget函数里根据目标位置与当前位置的差(wDiff)计算出需要的平均转速，也就是每一步的平均间隔时间(wAverageStepTime)，根据平均间隔时间设置TPM2C0V的增量(wTimerIncrement)，并启动定时器。在定时器的中断服务程序中驱动电机一步一步地向目标位置转动，当步进电机转动到目标位置后，让定时器停止工作。

另外，考虑到输入信号变化很快时，电机的转动速度可能跟不上信号的变化，从而在下一次测量完输入信号，对步进电机的目标位置再次进行设定时，电机还正在转动之中。这时，电机的转动速度有可能很快，在这种情况下如果重新启动定时器则有可能造成失步，因此SetStepperTarget程序根据电机当前的状态进行了分别的处理，其流程如图12所示。

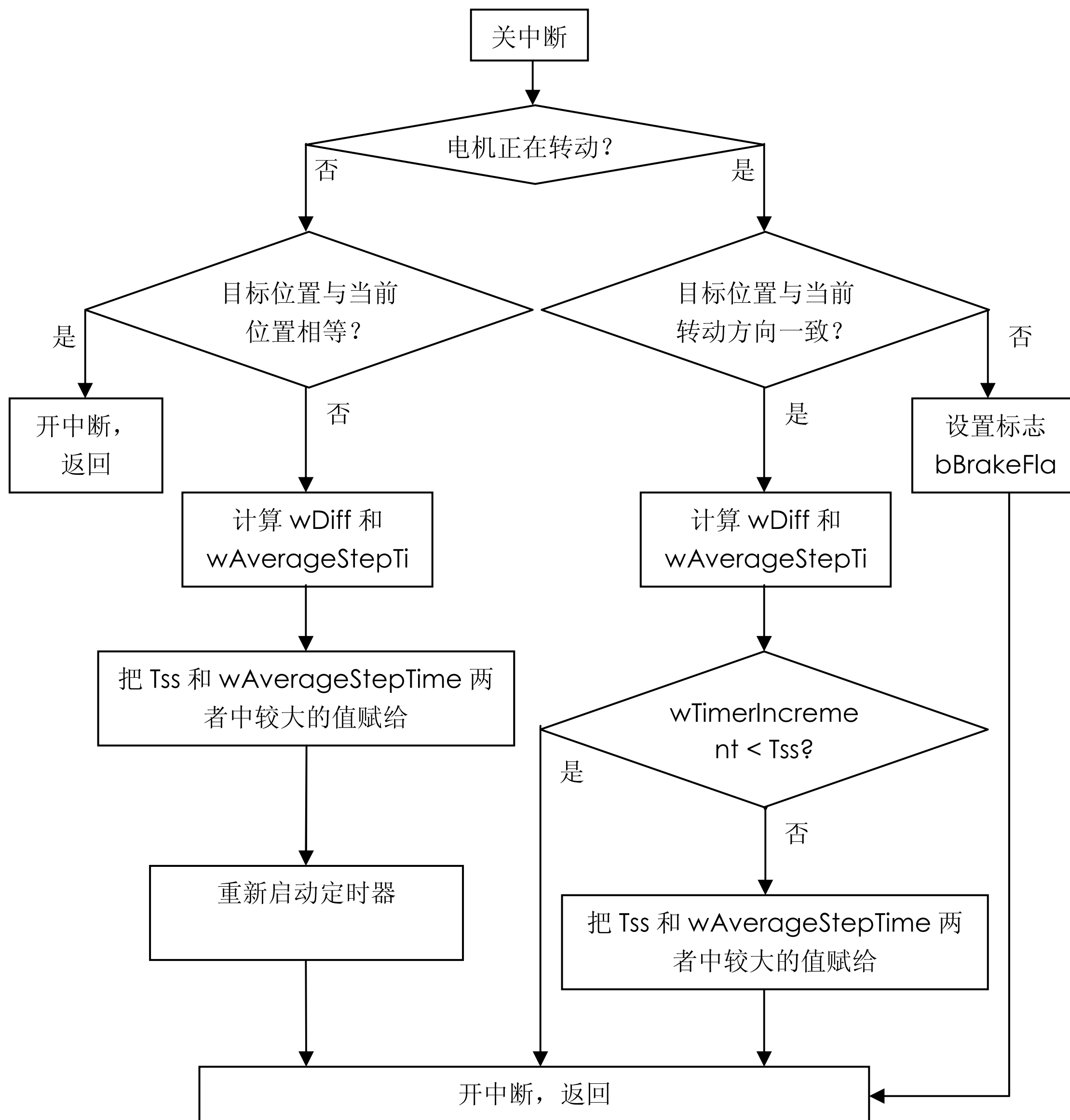

图12. SetStepperTarget()流程图

在定时器的中断服务程序中，首先判断bBrakeFlag标志，如果不是0，则表示当前转动方向跟目标位置不一致，需要先减速到停止，然后再反向转动。当bBrakeFlag为0时，根据平均间隔时间决定何时让电机转动一个分步/微步，以及是否需要进行加减速。当平均间隔时间大于Tss时，电机每隔一个平均间隔时间转动一步，作匀速转动。而当平均间隔时间小于Tss时，则让电机第一步的间隔时间为Tss，然后逐渐减小，即做加速转动，直到达到需要的平均转速或电机允许的最高转速；到接近目标位置时，则让每一步的间隔时间逐渐加长，电机减速转动，并且使最后一步的间隔时间不小于Tss，以保证不会失步。定时器中断服务程序的流程如图13所示。

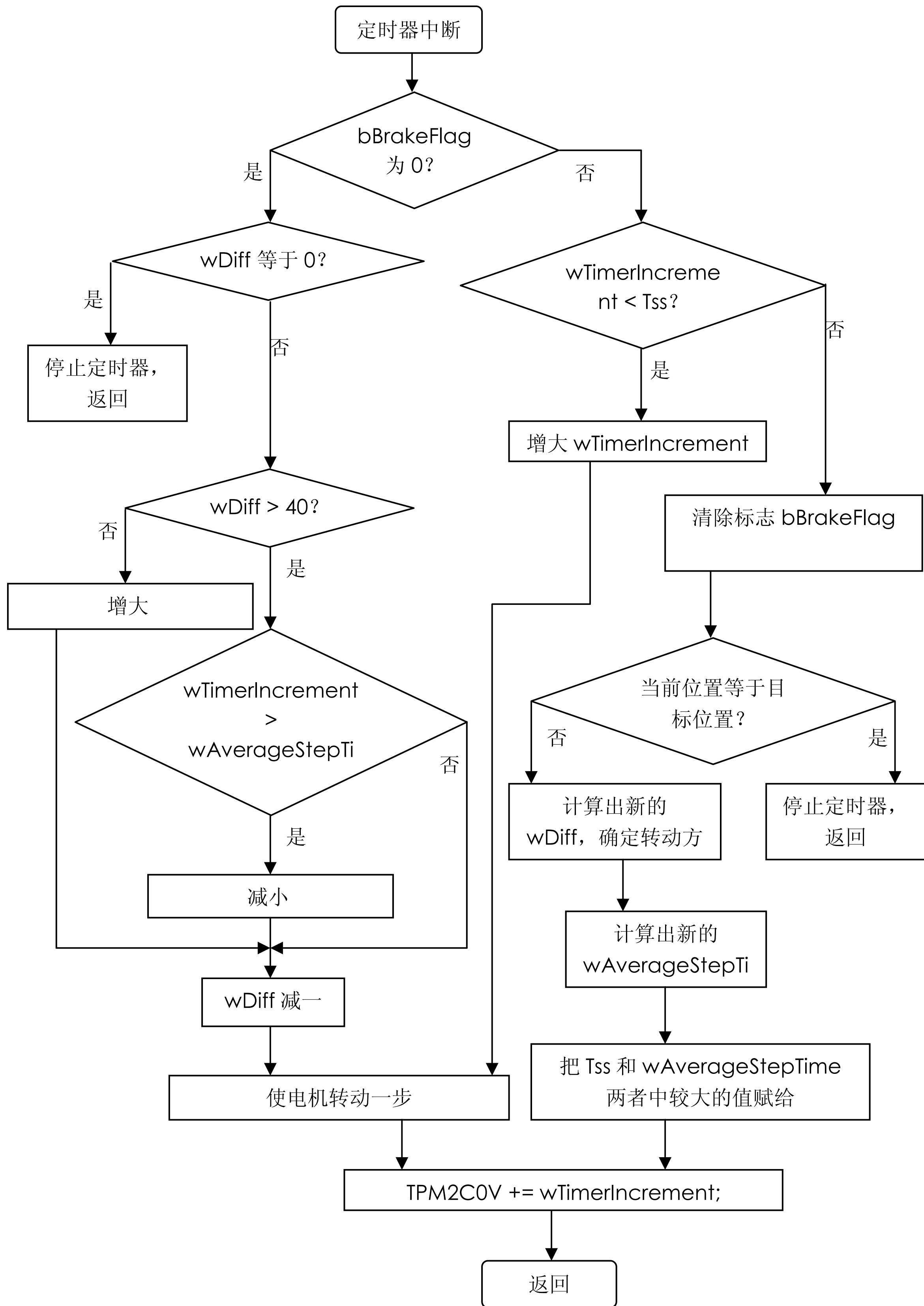

图13. 定时器中断服务程序流程图

总结

使用本文给出的驱动程序，用户的应用程序和驱动程序之间的接口非常简单，所有的控制步进电机的逻辑都封装在驱动程序中，而且驱动程序占用CPU的处理时间非常少。另外，本程序根据汽车仪表应用的特点对步进电机的转动速度进行了相应的控制，所以不管输入信号变化快慢，电机输出带动的指针都能快速、平稳地转动，以让人感觉很自然的方式将输入的物理量显示出来。